Entity-Based Access Control: supporting more expressive access control policies

Jasper Bogaerts, Maarten Decat, Bert Lagaisse, Wouter Joosen
iMinds-DistriNet KU Leuven, 3001 Leuven, Belgium
first.last@cs.kuleuven.be

ABSTRACT

Access control is an important part of security that restricts the actions that users can perform on resources. Policy models specify how these restrictions are formulated in policies. Over the last decades, we have seen several such models, including role-based access control and more recently, attribute-based access control. However, these models do not take into account the relationships between users, resources and entities and their corresponding properties. This limits the expressiveness of these models. In this work, we present Entity-Based Access Control (EBAC). EBAC introduces entities as a primary concept and takes into account both attributes and relationships to evaluate policies. In addition, we present Auctoritas. Auctoritas is a authorization system that provides a practical policy language and evaluation engine for EBAC. We find that EBAC increases the expressiveness of policies and fits the application domain well. Moreover, our evaluation shows that entity-based policies described in Auctoritas can be enforced with a low policy evaluation latency.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

General Terms

Management, Security, Languages

Keywords

Access Control, Access Control Model, Language, Attribute, Relationship, ABAC, Authorization, XACML, Policy Language, EBAC, Entity

1. INTRODUCTION

Access control is an important security measure that limits the actions that a user (typically referred to as the subject) can perform on resources, which are referred to as objects. This can typically be specified in externalized policies.

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
ACM 978-1-4503-3682-6/15/12 ...$15.00.
http://dx.doi.org/10.1145/2818000.2818008

Over the last decades, several access control models have been introduced to increase the expressiveness of these policies. One important such model is Attribute-Based Access Control (ABAC, [13, 24]). ABAC determines access based on attributes. Attributes are key-value pairs which are assigned to subjects, objects, actions and the environment. These attributes can then be compared with each other and literal values in order to determine if access is permitted. Such comparisons can also be combined into rules using logical connectives (i.e., *and*, *or*, *not*). As such, ABAC supports expressive policies.

However, a limitation of ABAC is that it cannot easily reflect relationships. This is evident in the following rule:

> *"A physician can view medical records that correspond to consultations for which the treating physician is of the same hospital, and corresponding to a patient that is also enrolled to the same hospital."* (A)

This rule implicitly evaluates the relationship of physicians with a patient's health record, and its translation to ABAC involves attributes such as *"object.consultation_physician_affiliation"* and *"object.consultation_patient_enrollments"*. Such synthetic attributes implicitly reflect the relationship that the object has with concepts in the application domain. Obtaining values for these attributes from the underlying data model requires specification of complex queries by a security expert. This may require considerable effort. Worse, some rules even force such queries to (partially) reflect the access control logic. For example, consider the rule:

> *"A physician can view a medical record of a patient if that physician had a consultation with that patient in the last year"* (B)

Here, the consultations in the last year could be represented by an attribute *subject.consultation_patients_last_year*. This attribute, however, already reflects rule (B). This introduces several disadvantages. First, it reduces readability of the policy. Second, it requires both the query as well as the policy to be modified whenever the rule changes (e.g., if consultations of the last two years are permitted). Third, since the query is specified in-code, modification of a policy may also require recompilation and redeployment of the application. This is a problem for applications that require continuous uptime, e.g., web applications.

In this paper, we introduce Entity-Based Access Control (EBAC). EBAC introduces the concept of entities as a first-class citizen in access control policies. Entities can have both attributes and relationships, which are addressed as key-value pairs. Similar to ABAC, EBAC compares at-

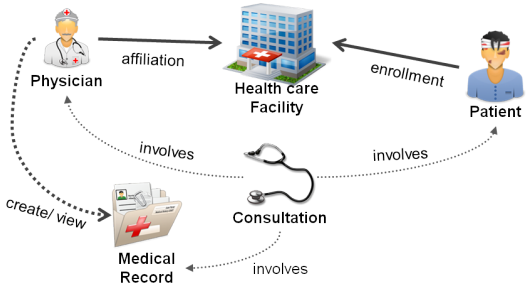


Figure 1: Physicians manage medical records. Patients are not primary actors of the application.

tributes with each other in order to make an access decision. However, EBAC policies can also navigate relationships and compare the attributes of related entities. Comparisons can be composed into complex expressions by means of logical connectives or predicate logic functions that can reason over relationships. Moreover, EBAC supports evaluation based on relationship paths of arbitrary length between entities and can reason about the entities along these paths. As a result, EBAC supports a more expressive way to reason about the application domain.

This paper introduces a policy model for EBAC. Moreover, it introduces *Auctoritas*, an authorization system that supports EBAC. Auctoritas includes a policy language and a policy evaluation engine. We evaluate what can be expressed in Auctoritas with regard to XACML and analyze the policy evaluation performance. We find that EBAC offers a more expressive model to specify policies in as compared to ABAC, and that Auctoritas policies can be enforced with low latency overhead.

The remainder of the paper is organized as follows: Section 2 analyses the problem in more detail. Section 3 elaborates on a EBAC policy model. Section 4 describes Auctoritas, an authorization system that supports EBAC. Section 5 evaluates the expressiveness of the Auctoritas policy language and the performance of its evaluation engine. Section 6 describes related work. Section 7 concludes the paper.

2. PROBLEM ANALYSIS

This section elaborates on the problem of limited expressiveness of ABAC in more detail. EBAC is motivated by rules that we encountered in case studies of electronic document processing [6], e-workforce management [7] and e-Health applications. For illustrative purposes, we focus on an example of an e-Health application that will be used throughout the paper. However, we do intend EBAC to be applicable to a wide range of applications.

2.1 Illustrative scenario

Consider an e-Health application that manages the medical records of patients for multiple health care facilities. The application enables physicians across health care facilities to look up and manage medical records for patients who participate in the system. Figure 1 illustrates the application.

From an access control perspective, medical records are the objects of the application. Medical records indicate the results of medical treatments and the actions that were performed during a consultation. Multiple medical records can be added for a single consultation. Patients can be enrolled to a health care facility to make this information available to all of its physicians.

1. A trainee physician can not create medical records.
2. A physician can view a medical record if the patient has granted his/her explicit consent to do so.
3. A physician can view a medical record if he/she is supervisor of the physician that created the record.
4. A physician can view a medical record if the corresponding patient had a consultation with him/her in the last year.
5. A physician can create a medical record if the categories he/she assigns to the record are all specializations of the physician.
6. A physician can create the medical record of a patient if that patient is enrolled to the same facility with which that physician is affiliated.
7. A physician can view a medical record if it corresponds to consultations for which the involved physician is from the same facility, and corresponds to a patient that is also enrolled to the same facility.
8. A trainee physician cannot view a medical record if it corresponds to consultations that took place more than four years before the trainee started.
9. A physician can view a medical record if the patient of its corresponding consultation already had a consultation with the physician's (in)direct supervisor.

Table 1: Example set of access control rules for a specific hospital that uses the application

Physicians are the primary users of the application. They are affiliated with a health care facility and have multiple specializations. They create and view medical records.

However, this needs to be constrained by access policies. Typically, health care facilities want to restrict access according to their own internal structure and requirements. Table 1 provides a set of rules that regulate access of physicians for the application from the case study. Evidently, this listing is meant to be illustrative and is not exhaustive. In the remainder of this section, we argue why most of these rules are too complex to express in ABAC. The next sections describe and illustrate how EBAC is able to express this.

2.2 Supporting technologies

Access control constraints are described in policies. Such policies can be separated from the application about which they reason, a technique commonly referred to as policy-based access control [20]. Externalizing policies from the application code offers several benefits. First, it enables separation of concerns between security experts and application developers. Second, it increases the modularity of both application code and security policies. Third, it supports policy adaptation without requiring application redeployment.

Policies adhere to a certain access control model. A popular access control model is Attribute-Based Access Control

(ABAC, [13, 24]). ABAC determines access by means of attributes. These attributes are represented as key-value pairs that are assigned to subjects, objects, actions and the environment. Whenever a subject attempts to access an object, a policy assessing the attributes of the subject, object, action and environment is evaluated.

Due to its widespread use, XACML [17] has become the de facto standard for specifying policies in ABAC. In XACML, attribute-based policies are structured as trees that contain a non-empty, ordered lists of rules[1]. Rules reason about access by means of logical expressions. They consist of a target, condition and an effect (i.e., `permit` or `deny`). Both target and condition are described by means of logical expressions that perform comparisons of attributes to each other or to concrete values. Expressions may also consist of logical connectives such as conjunction, disjunction and negation. If both target and condition evaluate to true, the rule evaluates to the specified effect. Otherwise, the rule is not applicable and does not influence the final access control decision. Policies also specify a target. If the target expression evaluates to true, the policy is further evaluated. Combination algorithms resolve conflicts that may arise whenever rules of a policy evaluate to different results. Common combination algorithms include `first applicable` (in which the first applicable rule dictates the outcome), `permit overrides` (in which a single permit decision overrides any other outcome) and `deny overrides`.

2.3 Problem elaboration

The state of the art supports the specification of expressive policies. However, not all rules of Table 1 can be expressed by current technologies. This is limited by what ABAC can specify.

One of the important drawbacks of ABAC is that it does not always apply seamlessly to the application domain about which it is supposed to reason. This is due to the fact that it only focuses on attributes that are assigned to either the subject, object, action or environment in its access rules. Moreover, ABAC does not properly support the expression of relationships. For example, an attribute that identifies the hospital of the physician that has created a medical record can be modeled as *"object.consultation_physician_affiliation_id"*. This requires the specification of a custom query to retrieve the values that correspond to this attribute, and involves manual mapping of attributes onto the data model by the security expert.

Such mapping is also an issue when translating complex relationships. For example, consider rule 9 from Table 1. A supervisor can be the direct supervisor of the subject, which can be modeled as the attribute *"subject.supervisor_id"*. However, an indirect supervisor cannot be represented in this way, and requires workarounds such as modeling an attribute *"subject.supervisor_ids"* which holds a set of all (in)direct supervisors of the subject. In order to accommodate such an attribute in the policy, the security expert needs to map a recursive relationship between physicians onto the data model. This can only be realized by means of a method which retrieves all indirect supervisors recursively.

Similarly, ABAC also does not support the specification of rules that reason about multiple attributes that are addressed over a relationship. For example, consider rule 4 from Table 1. This rule could be described using an attribute such as *"subject.record_ids_of_last_year"*, which returns all identifiers corresponding to consultations that the subject was involved in during the last year. However, a query to retrieve the values for this attribute at least partially reasons about the rule itself. This is clear when we want to change the rule to permit access to any medical records corresponding to consultations in the last *two* years. This would involve a modification of both the query as well as the attribute that is referred to in the access control policy. Moreover, such an attribute reduces the readability of the policy, and may require recompilation and redeployment of the application when the rule is modified.

As a result, ABAC lacks expressiveness as it does not support reasoning over relationships with entities other than subjects, objects and actions. This can become a problem when specifying complex rules over an application domain.

3. ENTITY-BASED ACCESS CONTROL

In order to cope with the drawbacks of ABAC, we present Entity-Based Access Control (EBAC). EBAC generalizes ABAC by introducing an entity-relationship model into the policy expressions. These expressions reason about relationships between entities next to supporting comparisons of entity attributes. EBAC integrates two core concepts:

ER model. The entity-relationship model (ER-model, [3]) is a data model for representing data of, among others, application domains. The ER-model combines three components: entities, their properties and inter-entity relationships. Properties assigned to entities are similar to the attributes of ABAC.

Logical expressions. Like XACML [17], EBAC supports policy assessments by means of evaluating logical expressions that compose rule conditions. These expressions compare properties with each other and with concrete values and can be combined by means of logical connectives (e.g., and, or, not). Additionally, EBAC supports predicate logic functions $\forall$ and $\exists$ and extends them to reason about relationships.

Note that EBAC also relates to the Relationship-Based Access Control (ReBAC, [12]) model. ReBAC is an access control model that focuses on privacy of end users in online social networks. To do so, ReBAC takes into account relationships between end users to determine access control decisions. However, it does not (or only limitedly [12]) support reasoning about properties of these end users and their relationships. This can be especially problematic when the policy needs to restrict access based on the values of certain properties (e.g., temporal attributes). Moreover, most ReBAC models do not take into account entities other than subjects and objects to specify the access control restrictions. Therefore, conceptually, this paper generalizes both ABAC and ReBAC into EBAC.

The remainder of this section introduces EBAC. First, it specifies how entities, attributes and relationships are modeled. Next, it elaborates on how expressions are represented.

3.1 Entity model and instantiation

In order to specify expressions, an *entity model* must exist that describes what can be compared in these expressions. EBAC supports comparison of entity attributes and reasons about relationships between entities.

[1]For brevity and because of its similarity to policies, we disregard policy sets for the description of XACML.

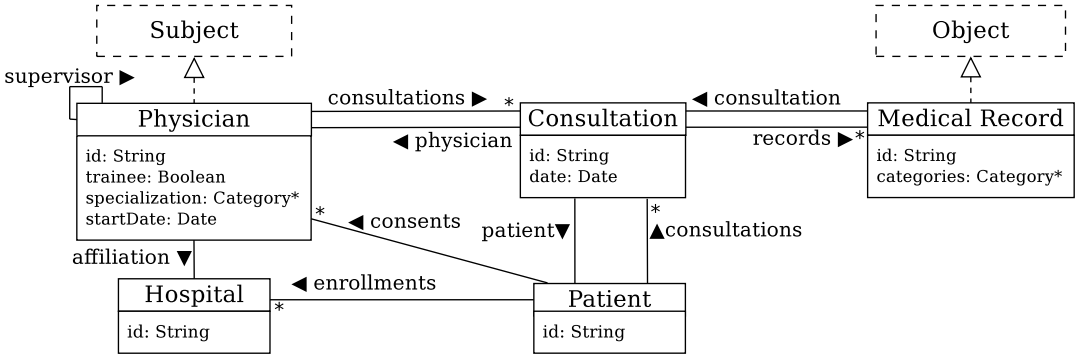


Figure 2: Entity model. For brevity, we omit action and environment entities, and relationship arities are not shown explicitly. Instead, a $*$ indicates a minimum arity of 0 and unbounded maximum. Otherwise, the relationship type occurs exactly once.

An entity model defines all types of entities, their attributes and the types of relationships that can exist between any two entity types for a certain application. This corresponds to the ER-model, which supports entity sets, their corresponding properties and relations to be specified [3]. Similar to relationship-based models [4], this takes into account entity types and their corresponding relationships. However, an entity model also takes into account attributes for the entity types and constrains the type and arity of both attributes and relationships. As a consequence, it enables the policy to correspond to the application domain because it can take into account any number of relationships and properties of any type for each relevant entity type.

An entity model constrains all expressions that can be specified in the policy for an application. In particular, it defines the entity types, their attributes and relationship types that can be specified in policy expressions. Moreover, it specifies data types so that restrictions can be imposed on comparisons between any two attributes. We define R as a finite set of restrictions that are imposed on relationships between two entity types. Such restrictions formalize the minimum and maximum number of relationships of a certain relationship type that are present in the instantiation of the entity model.

Entity graph. Entity and relationship types can be represented in a directed, multi-labeled graph $G(V, E)$ in which its vertices represent entity types and the edges represent relationship types. This is illustrated in an example entity model in Figure 2. This model is based on the illustrative example. Similar to [4], edges of G are labeled, supporting any relationship type between two types of entities. However, EBAC also supports multiple labels to be assigned to vertices in G. Consider L the set of all labels assigned to vertices $v \in V$. Set L contains tuples (v, k, t), in which

- Element $v \in V$ is the vertex to which the label is assigned.
- Element k represents a reference to the attribute.
- Element $t \in T$ identifies the data type of the attribute. We define T the set of all supported data types.

An entity model $S = (G(V,E),\ L,\ T,\ R)$ specifies the entities that can be evaluated in the expressions of a policy, together with their attributes and relationships. For the entity model represented in Figure 2, the set of labels L includes the attributes (e.g., *startDate*), the set of types T includes the supported data types (e.g., *Category*) and the set

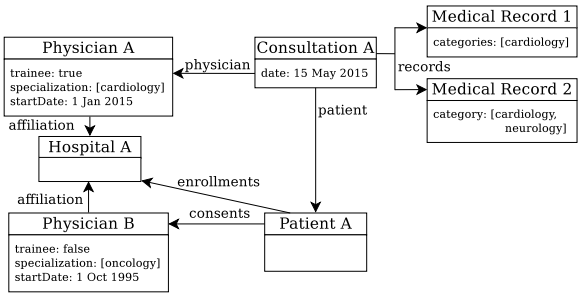


Figure 3: Example of an instantiation of the entity model that is illustrated in Figure 2.

R includes restrictions such as that the *affiliation* relationship is present precisely once for each physician. Informally, each entity type is identified by a set of outgoing edges of its corresponding vertex, their accompanying labels, and the attributes corresponding to the entity.

Instantiation. In order to evaluate a policy, values need to be retrieved from attributes of the relevant entities. The entities which are actually present for an application can be represented as an instantiation of the entity model. An example of such an instantiation is illustrated in Figure 3. An instantiation $G_I(V_I, E_I)$ reflects all instances of each entity type and their intermediate relationships. Also, the attributes *(v, k, t)* of the entity types are instantiated as tuples (i_v, k, val) in which:

- An entity $i_v \in V_I$ that corresponds to an entity type bound to a vertex $v \in V$ from the entity model.
- A key k that corresponds to the attribute in the model.
- A concrete value val of type $t \in T$ that was assigned to the attribute for this instance.

The model instantiation allows us to evaluate an access control policy for concrete entities. While such policies are specified in terms of the entity model, the evaluation deals with a concrete instantiation that takes into account the entities related to the relevant subject, object, action and environment over the pre-defined model.

3.2 Expressions

Expressions enable EBAC rules to support comparisons between attributes and values. Expressions only evaluate to `true` or `false`. EBAC supports three types of expressions: (1) simple expressions, which express a comparison

between two attributes; (2) composed expressions, which combine expressions with logical connectives (i.e., and, or, not); and (3) quantifier expressions, which introduce quantifier functions that reason about properties and relationship paths of arbitrary length. In order to address attributes in these expressions, we specify *path selectors*.

Path selectors

Access requests involve a *subject*, *object*, *action* and context in which it is performed (represented by the *environment*). Consequently, to retrieve attributes, they must be addressed starting from one of these four entities. However, as opposed to ABAC, EBAC supports traversing relationships of arbitrary length using path selectors.

Informally, a path selector refers to an attribute by specification of a path of relationship types, ending with an attribute of the final entity. For example, *'object.consultation.patient.id'* refers to the identifier that corresponds to the *patient* of the *consultation* to which the object relates.

More formally, consider S a finite set of instances of entities of a type in $\{subject, object, action, environment\}$. A *path selector* is a tuple (s, P, k) in which

- The entity instance $s \in S$ is the starting node from which the path is traversed.
- A finite, ordered list $P = (p_1, p_2, ..., p_n)$ contains the path of relationship types (each occurring exactly once) that is traversed to refer to the attribute. The elements must match the edge labels in the given order.
- The key of the attribute k that is queried from the addressed instance at the end of the path P.

For example, a path selector that is informally specified as *'object.consultation.patient.id'* is represented by $(medrec_1, (consultation, patient), id)$ for a medical record $medrec_1$.

Path selectors enable specification of attribute queries spanning over relationship paths. While we refer to attributes in the remainder of this paper, they are always referred by means of path selectors.

Simple Expressions

Simple expressions are the basic components of any composed expression. Informally, they compare two attributes or an attribute and a literal value. For example, consider:

$$\textit{'neurology' in object.consultation.physician.specialization}$$

This expression compares a concrete value (i.e., *'neurology'*) to a set of specialization values.

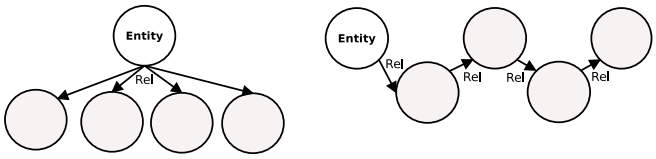
More formally, simple expressions are tuples $(l, r, \diamond)$ which:

- Elements l and r indicate the left and right leg of the comparison, respectively. The left leg is always an attribute, whereas the right leg can be either a literal value or an attribute. Note that attributes are specified by means of a path selector.
- Comparison operator $\diamond$ (e.g., $\leqslant$, $=$) indicates the comparison that will be performed between the elements.

Both l and r should be of the same type $t \in T$, or, alternatively, $\diamond$ is an operator that can handle a comparison between two elements of different types (e.g., a list type supports the comparison operator *in*).

The aforementioned expression is modeled as tuple:

$$((object, (consultation, physician), specialization), \textit{'neurology', in})$$



(a) Horizontal (b) Vertical

Figure 4: Evaluation of quantifier expressions. Note that horizontal expressions can also reason about multi-valued attributes, not depicted here.

In this example, the *in* comparison operator checks whether the category *'neurology'* is present in a list of categories that is retrieved by looking up the attribute for path selector *"object.consultation.physician.specialization"*.

Compound Expressions

Simple expressions can be combined using logical connectives into composed expressions. For example, consider:

$$subject.trainee \wedge object.consultation.date \leqslant env.now$$

This expression composes two simple expressions by means of a logical connective. A composed expression is either expression $\neg expr_1$ or a tuple $(expr_1, expr_2, conn)$ with

- Both $expr_1$ and $expr_2$ expressions.
- A logical connector $conn \in \{\wedge, \vee\}$.

Note that $expr_1$ and $expr_2$ can both be any type of expression. This supports composition of complex expressions.

Quantifier Expressions

Besides simple and compound expressions, EBAC also supports expressions that can reason about entities addressed by relationships. Two such expressions exist: *horizontal* and *vertical* quantifier expressions, as illustrated in Figure 4.

Horizontal quantifier expressions. Entity types can specify relationships that occur more than once. For example, a patient can designate an explicit consent to multiple physicians to view his/her medical records. Similarly, entities may have attributes with multiple values. For example, physicians can be assigned multiple specializations.

EBAC supports quantifiers to reason about entities that are addressed in these relationships. For example, consider rule 5 of Table 1. This rule includes the following condition:

$$\forall\ cat \in object.categories : cat \in subject.specialization$$

The rule reasons about a set of categories, and determines membership in *"subject.specialization"* for each value.

Similarly, EBAC supports reasoning about entities that are addressed with a relationship path with an arity higher than one. Consider rule 4 from Table 1. This rule essentially reasons about consultations and includes the condition:

$$\exists\ cons \in object.consultation.patient.consultations : cons.physician == subject \wedge cons.date < (env.now - 1\ year)$$

The above rule traverses the relationship path *"consultation.patient.consultations"* of the object and reasons about every entity that is addressed via this path.

In order to support this, EBAC makes use of *partial expressions* that are referred to inside the quantifier ($\forall$ or $\exists$). Partial expressions are expressions that contain *anonymous*

path selectors. Recall that normally, each path selector by which an attribute is selected includes a starting node. Such a starting node is an element of the set S (the set of all instances of entity types that represent a *subject*, *object*, *action* or *environment*). Contrarily, anonymous path selectors can have any instance as starting node. This enables partial expressions to be parameterized by combining the anonymous path selector with a path that starts from an element $i_v \in S$.

For example, the partial expression in the aforementioned rule is "$cons.physician == subject \wedge cons.date < (env.now - 1\ year)$". Because path selectors have a path starting at a subject, object, action or environment entity, the entity *"cons"* is an *anonymous* path selector. In order to reference attributes, they must be parameterized by every entity addressed by *"object.consultation.patient.consultations"*. This supports reasoning about entities using quantifiers.

More formally, we define *horizontal* quantifier expressions as tuples $(attr, pexpr,\ f\ \in\ \{\ \forall,\ \exists\ \})$ in which

- The element *attr* represents a sequence Seq of values, or entities that are addressed via a relationship with an arity greater than one.
- The element *pexpr* is a partial expression that is parameterized with each item $e \in Seq$. Note that anonymous path selectors must be of the same (entity) type than that of the elements $e \in Seq$.
- Function $f \in \{\forall, \exists\}$ indicates the quantifier type that is used for the expression.

Figure 4 (a) illustrates how an entity may relate to other entities in horizontal quantifier expressions.

Vertical quantifier expressions. Whereas horizontal quantifier expressions enable us to reason about relationships of a specific entity, they do not support reasoning about recursive relationships. Consider rule 9 from Table 1. This rule cannot be translated in terms of a *horizontal* quantifier expression. More specifically, its evaluation traverses the *"supervisor"* relationship of a physician until (a) a supervisor is found for which the condition is true, or (b) no more supervisor can be found for the relationship path.

To accommodate this, EBAC supports *vertical* quantifier expressions (shown in Figure 4 (b)). Vertical quantifier expressions are evaluated for each entity on a relationship path that is traversed recursively over an entity type. In the aforementioned rule, the relationship path is *"supervisor"*. Consequently, a condition is evaluated for each *physician* found while traversing this relationship.

More formally, we define a *vertical* quantifier expression as a tuple $(es, P, pexpr, f \in \{\forall_\rho, \exists_\rho\})$ in which

- Element *es* specifies a starting *entity* selector. Entity selectors, similar to path selectors, specify a path on which an entity is addressed. Contrary to path selectors, entity selectors do not address an attribute at the end of this path, but rather address the entity directly.
- The relationship path $P = (p_1, p_2, ..., p_n)$ a finite, non-empty, ordered list that specifies the path that must be traversed at each step in order to parameterize the partial expression *pexpr*. The last element p_n always ends at one or more entities of the same type as *es*.
- A partial expression *pexpr* that is parameterized with each entity along the path.
- Function $f \in \{\forall_\rho, \exists_\rho\}$, with $\forall_\rho$ the equivalent of universal quantifier $\forall$ and $\exists_\rho$ the equivalent of $\exists$.

Optionally, vertical quantifier expressions can be extended by $min, max \in \mathbb{N},\ min < max$ which indicate the minimal and maximal depth of the path for which the partial expression must hold.

Consider again the previously mentioned rule concerning the (in)direct supervisors. This rule contains the following vertical quantifier expression:

$$\exists_{\rho,supervisor}\ sv\ \in\ subject.supervisor :$$
$$\exists\ cons\ \in\ sv.consultations : (cons.patient == object.consultation.patient)$$

When evaluated, this rule traverses the subject's supervisor, evaluates the partial expression (starting from the *horizontal* quantifier expression). If no match is found, the same partial expression is evaluated for *subject.supervisor.supervisor*, and so on. This is repeated this until (a) there is no more supervisor, (b) the partial expression evaluates to `true`, or (c) a previously encountered physician is found.

While the *subject.supervisor* relationship has an arity of 1, EBAC also supports relationships with a higher arity. In this case, each entity from the relationship is used as a parameter for the partial expression. If no entity is found that leads to a definite answer for the evaluation, each entity's children are evaluated in a breadth-first manner. Next, the processes is repeated for each element until no new entities can be found. Duplicate entities are not revisited.

4. THE AUCTORITAS PROTOTYPE

To validate EBAC, we have incepted a practical prototype called Auctoritas. Auctoritas is an authorization system that supports EBAC. Auctoritas includes a policy language and evaluation engine. The prototype is available online[2].

The STAPL [15] authorization system served as a basis for the Auctoritas implementation. STAPL focuses on modular policy specification, supporting attribute-based policies.

4.1 Policy language

The Auctoritas policy language was incepted as a DSL in Scala, and has a similar structure to XACML [17]. More specifically, Auctoritas is also a policy-based language that supports comparing attributes with each other and concrete values by means of expressions. Such expressions can be composed using logical connectives (e.g., and, or, not) and are evaluated to determine access. Like XACML, Auctoritas policies are organized in policy trees that use combination algorithms (e.g., `deny overrides`) to resolve evaluation conflicts. A rule in Auctoritas can either evaluate to `permit`, `deny` or `not applicable`. However, as opposed to XACML, Auctoritas is entity-based.

Listing 1 illustrates the translation of all rules from Table 1 as rules in a single policy. Each rule can specify a condition and effect. For example, rule 1 in Listing 1 specifies a `deny` decision whenever the subject is a trainee (i.e., the value of the *trainee* attribute is `true`). Policies can also specify a target and combination algorithm. For example, Listing 1 only regards requests to view medical records[3]. If this is not the case, the policy is `not applicable`. Similarly, a rule evaluates to `not applicable` when its condition expression evaluates to `false`.

[2] `http://people.cs.kuleuven.be/~jasper.bogaerts/auctoritas/`

[3] This is a simplified version of the rules in Table 1. However, for the evaluation in Section 5, we did specify them in compliance to Table 1.

```
Policy("Policy") := when (action.id === "view" & resource.type_ === "MedicalRecord") apply FirstApplicable to (
 Rule("rule_1") := deny iff (subject.trainee)
 Rule("rule_2") := permit iff (subject.id in object.consultation.patient.consent)
 Rule("rule_3") := permit iff (subject.id === object.consultation.physician.supervisor.id)
 Rule("rule_4") := permit iff (subject.consultations.exists(
                   consultation => consultation.patient.id === object.consultation.patient.id &
                      environment.now <= (consultation.date + 1.years)))
 Rule("rule_5") := permit iff (object.consultation.categories.forall(cat => cat in subject.specializations))
 Rule("rule_6") := permit iff (subject.affiliation in object.consultation.patient.enrollments)
 Rule("rule_7") := permit iff (object.consultation.physician.affiliation.id === subject.affiliation.id &
                               (subject.affiliation in object.consultation.patient.enrollments))
 Rule("rule_8") := deny iff (subject.trainee & subject.startDate > (object.consultation.date + 4.years))
 Rule("rule_9") := permit iff (subject.supervisor.existsOnPath(supervisor => object.consultation
                              .patient.consultations.exists(cons => cons.physician.id === supervisor.id)))
)
```

Listing 1: Translation of all rules in the Auctoritas policy language. Objects are referred to as resources. For brevity, we have included the rules in a single policy and specified only the *view* action. However, Auctoritas also supports policy trees.

Auctoritas also supports path selectors to traverse an arbitrary number of relationships. Consider rule 3 in Listing 1. Starting from an object, this rule traverses the *consultation*, *physician* and *supervisor* relationships, respectively, to access the *id* attribute. If any of these relationships is not present, the expression evaluates to `false`. Note that the traversed relationship types in this example all have a maximum arity of 1. If a path selector contains a relationship type with a higher maximum arity, more elaborate logical functions are required.

This is accommodated by quantifier expressions. Consider rule 4 in Listing 1. This rule is supported by a horizontal quantifier expression that reasons about every consultation that corresponds to the subject. The quantifier assesses each consultation and uses it to parameterize the inner (partial) expression, which looks up the *patient* and *date* corresponding to the consultation. This inner expression is specified as a lambda function that in practice assesses each identifier corresponding to the relationship it parameterizes. For example, rule 4 first looks up a list of all identifiers that correspond to the consultations of the subject, and for each consultation, looks up further attributes and relationships that correspond to a consultation by providing the identifier and evaluating the expression. Note that while rule 5 in Listing 1 also uses a horizontal quantifier expression, the expression only deals with a list of values as opposed to entities. Consequently, it does not require elaborate evaluation techniques.

A approach similar to the evaluation of rule 4 is taken to evaluate vertical quantifier expressions, such as the condition of rule 9 in Listing 1. This approach requires a minimal amount of attributes to determine access. Alternatively, prefetching the required attributes for each entity on the path could reduce the policy evaluation overhead. This is an interesting topic for future research.

In order to specify access control policies, the security expert first sets up an entity model that describes the concepts that will be reasoned about in the policy, and maps them to a database where they are retrieved. Such specification is based on the application model. Hence, it can be automated. This constitutes an interesting topic for future work.

4.2 Supporting infrastructure

In [23], Vollbrecht et al. describe a reference architecture for policy-based authorization systems. This architecture has been adopted by many authorization systems, among others XACML [17]. It consists of several components such as a Policy Enforcement Point (PEP) that enforces an authorization decision, a Policy Decision Point (PDP) that evaluates access control policies to make a decision and a Policy Information Point (PIP) that obtains the information required to evaluate a policy (e.g., attributes).

Auctoritas also adopts this architecture. More specifically, it adopts the interfaces of each of the components in order to retain compatibility. However, Auctoritas also extends the functionality of the PDP and PIP in order to cope with the differences between EBAC and other models. Such an extension can introduce latency overhead during policy evaluation. This is evaluated in Section 5.

The PDP is extended to support relationships. This also includes elaborate evaluation logic for horizontal and vertical quantifier expressions. The PIP is extended to handle path selectors and requests for attributes parameterized in anonymous path selectors. It obtains the value(s) for the attribute, addressed by a path selector, by automatically constructing a query that traverses the path by means of *join* operations. For example, the PIP handles *"object.consultation.patient.id"* by composing a query that joins the *medical record*, *consultation* and *patient* together based on their relating attributes. The *id* attribute is selected from this join query.

5. EVALUATION

Section 4 demonstrated the applicability of EBAC and introduced Auctoritas. This section evaluates both the expressiveness and policy evaluation performance overhead of Auctoritas with regard to XACML.

5.1 Expressiveness

In order to compare the expressiveness of ABAC and EBAC, we have translated the rules that were introduced in the illustrative example of Section 2 into XACML and Auctoritas. For Auctoritas, this was illustrated previously in Listing 1.

Table 2 provides a comparison of expressiveness of XACML and Auctoritas with regard to the rules that were specified in the illustrative example. Table 2 indicates categories of expressions that can be specified in Auctoritas, respectively XACML, with a ✓ if this does not involve changing the PIP (e.g., by providing custom queries or access logic on the returned attributes). Expression categories that cannot be expressed in this way are indicated with a ×. We assume

Category	XACML	Auctoritas	Rules
Regular attribute comparisons	✓	✓	1
Composed relationships	×	✓	2, 3, 6, 7, 8
Simple horizontal quantification	✓	✓	5
Relationship horizontal quantification	×	✓	4
Vertical quantification	×	✓	9

Table 2: Comparison of expressiveness of XACML and Auctoritas with regard to the rules specified in Section 2. XACML supports the first and third category, and as a result, can express rules 1 and 4 from the illustrative example. Auctoritas supports all categories of expressions. Hence, Auctoritas can express all rules from the illustrative example without additional configuration of the PIP. A translation of all rules to Auctoritas is illustrated in Listing 1.

the underlying data model for the attributes to coincide with the entity model that was presented in Figure 2.

Note that all presented rules can be specified in XACML. However, as indicated earlier, complex rules, particularly involving relationships, require additional implementation in the PIP. This decreases readability of the policy. Moreover, it may require both the attribute and query to be reconfigured whenever a rule is modified. Rule modification may even require recompilation and redeployment of the PIP.

The first row of Table 2 categorizes the expressions that can be described using regular attribute comparisons. For example, the rule *"A trainee physician can not create medical records"* contains a condition that only addresses attributes of a subject, object, and action directly. This expression category is supported by both XACML and Auctoritas.

However, as XACML cannot examine relationships directly, the expression category concerning composed relationships is not supported. This category comprises comparisons that use path selectors which follow relationships to address attributes. For example, path selector *"object.consultation.physician.trainee"* traverses two relationships before addressing an attribute. Such path selectors generally involve various look-ups which, for XACML, requires *join* statements in the underlying customized query for attributes maintained in a database. Therefore, as opposed to Auctoritas, XACML requires PIP changes to support this.

Similar to XACML, Auctoritas supports simple quantifications. This category involves horizontal quantifier expressions that only compare an attribute to all elements of a sequence, or compare all elements of two sequences with each other. For example, an expression may evaluate whether a value is greater than each element in a sequence. XACML provides functions such as `any-of`, `all-of` and `any-of-any` to accommodate such expressions. However, XACML does not support expressions such as *"A physician can view medical records of patients who had a consultation with him/her in the last year"*. This rule involves two separate expressions: (a) patients should have had a consultation with the physician; and (b) that consultation should have occurred in the last year. These expressions must have a correlating entity (i.e., the consultation), and therefore, cannot be evaluated separately. Hence, such rules cannot be specified in XACML without implementing specialized queries at the PIP.

Lastly, XACML does not support vertically quantified expressions on relationships. For instance, the rule *"A physician can view a medical record if the patient of its corresponding consultation has previously had a consultation with the acting physician's (in)direct supervisor"* (rule 9) requires a recursion over all supervisor relationships that start from the subject. In order to support this, XACML requires the PIP to accommodate a method that performs this recursion to generate the data set on which the rule can be evaluated. Consequently, this also requires customization at the PIP in order to support translation in XACML.

5.2 Evaluation of performance overhead

Complex evaluation over relationships requires additional evaluation logic in both PIP and PDP components of the evaluation engine. This introduces policy evaluation overhead. In order to measure the overhead induced by the evaluation, we analyze the performance and compare it to XACML. Our analysis is based on the translation of the rules from Table 1. When the rules were not directly supported in XACML, we configured the XACML PIP to perform queries to enable the translation of the rule. For example, a path selector *"object.consultation.patient.id"* in Auctoritas is represented as an attribute *"object.consultation_patient_id"* in XACML. Note that this can lead to problems, as was indicated earlier in this paper.

Setup. Each rule was organized in a separate policy for which the evaluation times were measured for both languages. We did not measure the evaluation overhead of policies with multiple rules, nor did we take into account the parsing overhead of the policies (a known problem for XACML [21]). Rather, we focused on the overhead introduced by attribute look-ups and expression evaluation. Both PIPs used JPA[4] to communicate with the database containing the attributes. We disabled all data caching for both PIPs. Requests to the PIPs occurred using local TCP requests via Apache Thrift[5]. We used the SunXACML implementation[6] for the XACML evaluation engine. The evaluation was performed on a Intel(R) Core(TM) i5-3340M CPU @ 2.70GHz with 8GB RAM running Fedora 20. Each rule was evaluated for 100000 applicable requests (i.e., the target matched, and as a result, attributes needed to be fetched). The results omit the 2.5% minimal and maximal outliers.

Results. Figure 5 illustrates the results of the evaluation. This figure shows that for most rules, Auctoritas performs significantly better than XACML.

For regular attribute comparison rules, the mean difference is 0.4ms in favor of Auctoritas. The simplified structure of Auctoritas as compared to the SunXACML implementation reduces its evaluation times.

In addition, composed relationships (i.e., rules 2, 3, 6, 7

[4]See `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`

[5]Available at `https://thrift.apache.org/`

[6]Available at `http://sunxacml.sourceforge.net/`

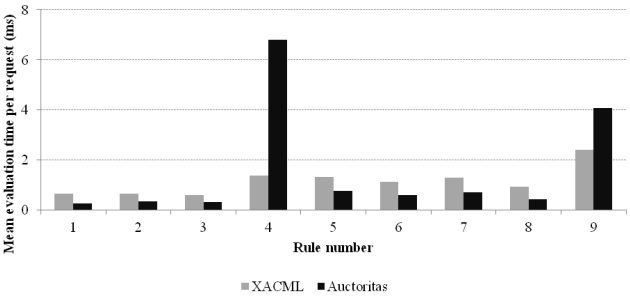


Figure 5: Evaluation overhead comparison between XACML and Auctoritas. EBAC evaluation performs slightly better. Rules 4 and 9 introduce overhead due to the number of separate attribute look-ups required to evaluate the rules.

and 8) evaluate significantly faster than the XACML implementation. This indicates that the number of relationships on a path does not have a great influence on the evaluation time. For instance, rule 7 has eight relationships that are traversed, but does not involve significant overhead compared to rule 3, which requires only three relationship traversals. Auctoritas automatically generates queries to accommodate relationship traversals via joins. This reduces the evaluation time (e.g., separately looking up each entity on a path for rule 3 resulted in a mean evaluation time of 1.58ms compared to 0.3ms).

However, Figure 5 also shows that rules 4 and 9 perform better in XACML. These reflect the relationship horizontal quantification and vertical quantification expressions, and their evaluation introduces a larger overhead. For the horizontal quantification (i.e., rule 4), the overhead as compared to XACML is 5.4ms per request. In the prototype implementation, each parameterization of the expression involves a separate look-up for each entity. This introduces significant overhead compared to a query that filters out attribute values of entities that do not need to be evaluated, as in our XACML translation. Note that the evaluated rule is inherently an expensive operation that may involve many consultation entities[7]. Evidently, the evaluation time will rise with larger data sets, and therefore such rules must be cautiously specified. The optimization for this is an interesting topic for future work, and may involve performing queries to the PIP that retrieve the attribute values for each of the entities that will be parameterized in a single request.

A similar issue occurs for vertical quantification (rule 9), where there is an overhead of 1.7ms compared to XACML. Optimization for this may involve additional logic to be added to the PIP to cope with the number of look-ups.

Discussion. Although Auctoritas does introduce overhead for complex rules compared to XACML, it does not require security experts to implement any queries or specialized lookup methods at the PIP. Such implementation could be considered a more expensive operation, as it may introduce errors. Moreover, it could require considerable effort for the security expert to implement such queries. To realize the translation in XACML, we have specified a total of eighteen queries (twelve of which required at least some form of table joining and filtering logic to be performed in the database) and one method (to realize the vertical quantification). For large policies, such configuration effort is likely to grow. Auctoritas provides a technology to reduce this effort while reasoning about the application domain.

[7]In this evaluation, the mean size of the operative data set was 23 consultations.

6. RELATED WORK

Over the last decade, various models have been proposed to mitigate issues of RBAC [8]. In this regard, this work is primarily influenced by Attribute-Based Access Control (ABAC) and Relationship-Based Access Control (ReBAC).

ABAC [13, 24] introduced the use of attributes in order to determine whether access is permitted. While this supports expressive policies, it does not allow specification of complex relationships. One important supporting technology is XACML [17]. XACML introduces an attribute-based policy language that also supports policy trees and obligations. These aspects have drawn a lot of interest (among others, [16, 18]). However, these works mostly focus on formalization or analysis of combination algorithms. Our work focuses on the expressiveness of attribute-based policies in XACML, as compared to Auctoritas.

Relationship-Based Access Control (ReBAC, [4, 9]) takes into account relationships in order to make access control decisions. This supports the expression of a great deal of rules. ReBAC has typically been applied to social networks [2, 12, 14]. However, Crampton et al. [4] introduced a work in which they (a) argue a wider applicability of relationships than only social networks, and (b) consider entities other than users to be relevant in specifying relationships.

Carminati et al. [2] analyzed the requirements with regard to access control for social networks and introduced a model which constrained access based on the type, depth and trust level of relationships and provided client-side enforcement of access control according to a rule-based approach. This work takes into account depth and type of relationships to determine access, which is also adopted in our work.

Giunchiglia et al. proposed RelBAC [10], a model that is similar to ReBAC. Like our work, the authors associate RelBAC to the Entity-Relationship model [3].

In [9], Fong identifies, among others, the concept of composable relations as a building block to build complex relationships based on more primitive relationships. This is related to our concept of vertical quantifier expressions.

While ReBAC is an interesting access control model that has influenced this work, a general issue for ReBAC is that it does not directly support comparisons of properties of the involved entities. As a consequence, it does not always map seamlessly onto the application domain.

Ribeiro et al. introduced the SPL policy language [19], which supports entities and their relations, comparison of properties and quantifiers. Ruby CanCan [1] is a policy language that has a lot of similarities with SPL. Among others, it also supports addressing of entities and attribute comparisons and is integrated seamlessly with the application model. Hachem et al. [11] introduced a policy framework for controlling access in mobile applications. This work featured, among others, a policy language which supported expressive policies. While these policy languages support reasoning about entities and relationships in a natural way, they do not support vertical quantifier expressions, and hence, do not entirely support EBAC. Moreover, they do not support policies to be organized in trees.

Also relevant is the work of Verhanneman et al. [22]. Ver-

hanneman et al. enforce separation of concerns by supporting organization-wide policies for which the rules are enforced in an aspect-oriented framework. This enables the security expert to reason about application-specific concepts using Java code. However, changing such a policy requires recompiling the application code. Separation of concerns was also addressed by De Win et al. [5], which motivated the use of aspect-oriented programming to enforce it.

7. CONCLUSION

In this paper, we have presented Entity-Based Access Control (EBAC), an access control model that supports both comparison of attribute values, as well as traversing relationships of arbitrary entities. EBAC maps onto the application domain in a expressive way because its underlying ER-model can be applied to most application models. We have proposed a model and validated that it is feasible to support EBAC by means of the Auctoritas policy language. The evaluation of Auctoritas indicates that (a) it supports relationships in a more expressive way than XACML and (b) that it introduces only limited performance overhead for policy evaluation. Support for these expressions involves no custom query specifications and hence reduce the management effort involved when such rules must be supported. Consequently, we are convinced that EBAC offers a promising new approach for access control.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

8. REFERENCES

[1] R. Bates. Ruby CanCan. `http://www.rubydoc.info/gems/cancan/`, 2015. [Online; accessed 4-February-2015].

[2] B. Carminati, E. Ferrari, and A. Perego. Rule-based access control for social networks. In *OTM Meaningful Internet Systems 2006*. Springer, 2006.

[3] P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM TODS*, 1(1), 1976.

[4] J. Crampton and J. Sellwood. Path conditions and principal matching: a new approach to access control. In *ACM SACMAT*, 2014.

[5] B. De Win, F. Piessens, W. Joosen, and T. Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Workshop on the Application of Engineering Principles to System Security Design*, pages 1–10, 2002.

[6] M. Decat, J. Bogaerts, B. Lagaisse, and W. Joosen. The e-document case study: functional analysis and access control requirements. Technical report, KU Leuven, 2014.

[7] M. Decat, J. Bogaerts, B. Lagaisse, and W. Joosen. The workforce management case study: functional analysis and access control requirements. Technical report, KU Leuven, 2014.

[8] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-based Access Control. *ACM TISS*, 2001.

[9] P. W. Fong. Relationship-Based Access Control: Protection Model and Policy Language. In *ACM Data and application security and privacy*, 2011.

[10] F. Giunchiglia, R. Zhang, and B. Crispo. Relbac: Relation based access control. In *Semantics, Knowledge and Grid, 2008*. IEEE, 2008.

[11] S. Hachem, A. Toninelli, A. Pathak, and V. Issarny. Policy-based access control in mobile social ecosystems. In *Policies for Distributed Systems and Networks (POLICY)*. IEEE, 2011.

[12] H. Hu, G.-J. Ahn, and J. Jorgensen. Multiparty access control for online social networks: model and mechanisms. *IEEE Knowledge and Data Engineering*, 25(7), 2013.

[13] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, 2014.

[14] S. R. Kruk, S. Grzonkowski, A. Gzella, T. Woroniecki, and H.-C. Choi. D-FOAF: Distributed identity management with access rights delegation. In *The Semantic Web–ASWC 2006*. Springer, 2006.

[15] J. Moeys and M. Decat. Simple Tree-structured Attribute-based Policy Language (STAPL). `https://github.com/stapl-dsl`, 2015. [Online; accessed 23-September-2015].

[16] Q. Ni and E. Bertino. xfACL: An extensible functional language for access control. In *Proceedings of the 16th ACM SACMAT*. ACM, 2011.

[17] OASIS. eXtensible Access Control Markup Language (XACML) Standard, Version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf`, 2013.

[18] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson. The logic of XACML. In *Formal Aspects of Component Software*. Springer, 2012.

[19] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An Access Control Language for Security Policies and Complex Constraints. In *NDSS*, volume 1, 2001.

[20] P. Samarati and S. Vimercati. Access Control: Policies, Models, and Mechanisms. In *Foundations of Security Analysis and Design*, pages 137–196. 2001.

[21] F. Turkmen and B. Crispo. Performance evaluation of XACML PDP implementations. In *Proceedings of the 2008 ACM workshop on Secure web services*, 2008.

[22] T. Verhanneman, F. Piessens, B. D. Win, and W. Joosen. Uniform application-level access control enforcement of organizationwide policies. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[23] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. RFC 2904: AAA Authorization Framework, August 2000.

[24] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *Proceedings of IEEE International Conference on ICWS*, 2005.